**Design & Analysis of Algorithms Programming Project**

**Comparison of three different algorithm’s time complexities**

**By Tommy Las**

**Problem Definition:**

Get the ith order statistic from different size n arrays using three different algorithms with different complexities. Insertion Sort Algorithm uses an approach that iterates through the array, exchanging the key with the previous element if key < previous element. Heapsort Algorithm sorts the array using the heap data structure, following the heap property. Both are sorting algorithms, and we return the ith order statistic from the sorted array. Randomized select gets ith order statistic by randomly partitioning the array and returning the pivot, divide and conquer approach. The program runs each algorithm for each array of size n = 1.000 to 10.000, five times for each array size n and gets the average running time in microseconds. The purpose of this project is to compare complexities n2, n log (n), and n.

**Algorithms and RT Analysis:**

**Insertion Sort Algorithm RT = O(n2):**

**INSERTION-SORT(A, index) // for A[0..n-1]**

n = A.length

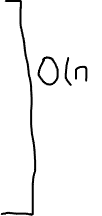
for j = 1 to n-1

key = A[j]

i = j – 1



while i >= 0 and A[i] > key



A[i+1] = A[i]



i = i - 1

A[i+1] = key

**Heapsort Algorithm RT = O(n lg(n) ): // for A[0..n-1]**

**PARENT(i):**



Return floor(i/2)

**LEFT(i):**



Return 2i + 1



**RIGHT(i):**



Return 2i +2

**MAX-HEAPIFY(A, i, n, heap\_size):**

n = heap\_size

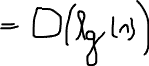
l = left(i)

r = right(i)

if l < n and A[l] > A[i]



largest = l



else:



largest = i

if r < n and A[r] > A[largest]:

largest = r

if largest != i:

exchange A[i] with A[largest]

MAX\_HEAPIFY(A, largest, n, heap\_size)

**HEAPSORT(A):**

n = A.heap\_size = A.lenght

BUILD\_MAX\_HEAP(A, n, heap\_size)



for I = n-1 downto 1

exchange A[0] with A[i]



A.heap\_size = A.heap\_size -1



MAX HEAPIFY(A, 0, n, heap\_size)



**BUILD-MAX-HEAP(A, n, heap\_size):**

for i = floor(n/2) downto 0



max-heapify(A, I, n, heap\_size)

**Randomized Select Algorithm RT = O(n): // for A[0..n-1]**

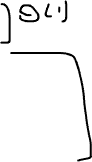
**PARTITION(A, p, r):**

x = A[r]

i = p-1



for j = p to r-1



if A[j] <= x:



i = i + 1

exchange A[i] with A[j]

exchange A[i+1] with A[r]



return i+1

**RANDOMIZED-PARTITION (A, p, r):**

i = random(p, r)

exchange A[r] with A[i]



return partition(A, p, r)



**RANDOMIZED-SELECT(A, p, r, i):**

if p == r:



return A[p]



q = RANDOMIZED-PARTITION(A, p-1, r-1)

k = q - p + 1

if i == k:



return A[q]

elif i < k:

return RANDOMIZED-SELECT (A, p, q-1, i)

else: # i > k

return RANDOMIZED-SELECT (A, q+1, r, i-k)

**Experimental Results:**

**Table

Description automatically generated**

**Table

Description automatically generated**

**Table

Description automatically generated**

**Chart, line chart

Description automatically generated**

**For the graph above, the range for y is 0 msec to 250msec, so we can look and compare the lines better**

Chart, line chart

Description automatically generated

Chart, line chart

Description automatically generated

Chart, line chart

Description automatically generated

**Experimental Results:**

This assignment was extremely helpful and interesting, because we can see more in depth and experiment with running time complexities of n2, n log(n), and n. The first project requirements, n had values from 10,000 to 200,000. For insertion sort algorithm, it would take seconds, couples of minutes to get the average running time, the program would be running for 2-3 hours and it only got to n = 120,000. But with the other two algorithms, Heapsort and Randomized Select, would take a couple microseconds, but the difference was very huge. This surprised me and made me realize how crucial the time complexity of algorithms is. When the requirements were changed, I had n values from 1,000 to 10,000, and I was able to run until the end the Insertion Sort Algorithm. The difference was still huge, comparing Insertion Sort with Heapsort and Randomized Select. If we look at graph #1, the one that compares the Empirical Values of the three algorithms, we can look at the differences of running times, where n2 would increase very sharply, n log(n) has a decent running time, but we can see how n is really close to y = 0 and is the most efficient one. For the comparison of Theoretical RT and Empirical RT for Insertion Sort and Heapsort algorithms are consistent. For randomized select the Theoretical RT and Empirical RT is not consistent, which is logical, because the randomized select uses an approach with randomized numbers using divide and conquer. The RT for the best-case is Θ(n) and worst-case RT is Θ(n2), and since we ran each n size array 5 times, the average for example, could be less than the previous n size values.

I was not aware how important time complexity was. This class and project made me more knowledgeable of algorithms and its complexities. From now on and later in my career, I will pay more attention on the algorithms I use and be mindful of its running time.

**References:**

* Introduction to Algorithms, 3rd edition, by T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, The MIT Press, 2009, ISBN: 0262033844.
* GeekForGeeks: For some help with algorithms, when switching to starting index 0. All algorithms follow pseudocode from the book, with its respective changes due to starting index 0.